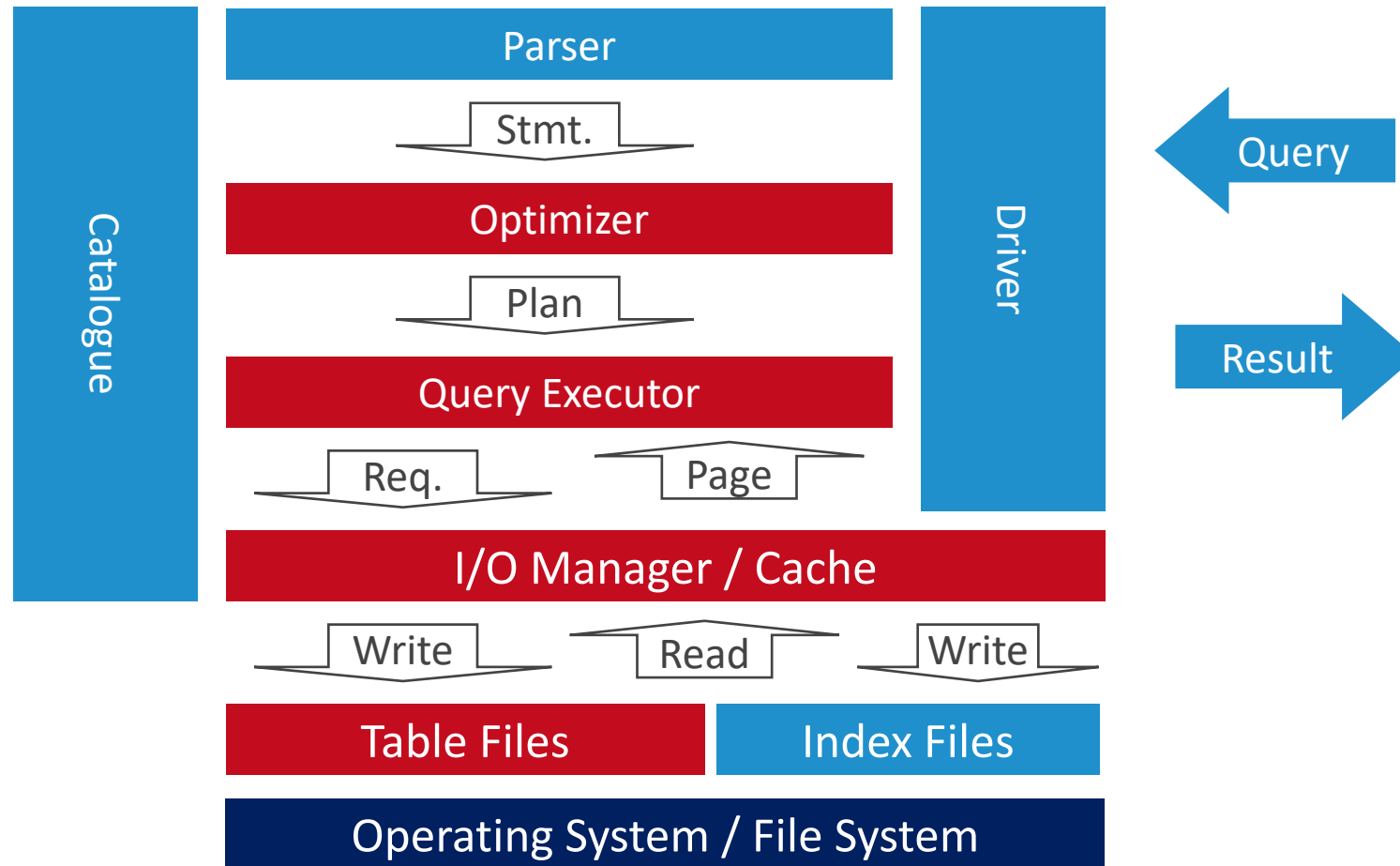


DBTLAB – Exercise 6: Query Execution - Operators III

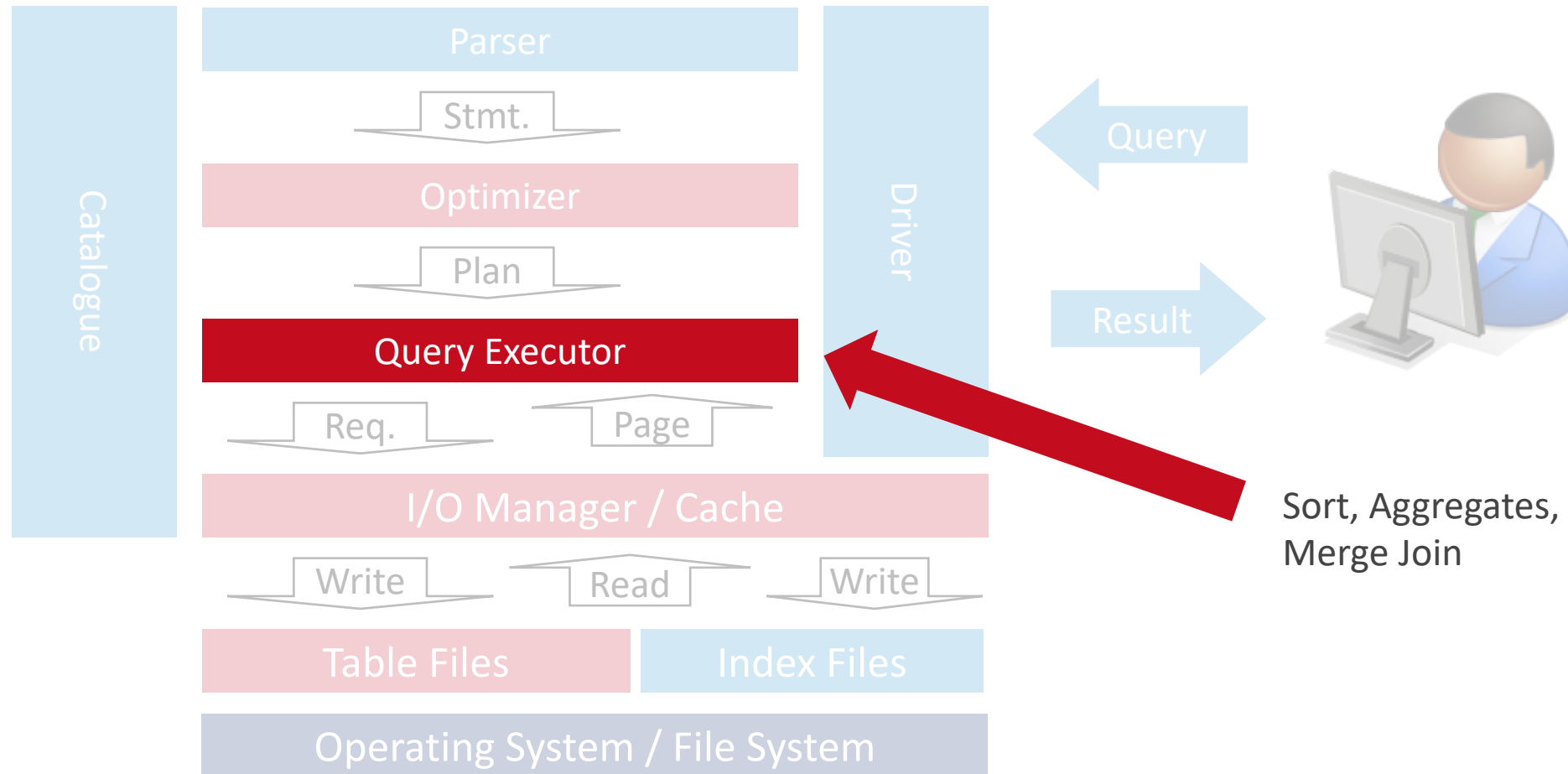
Rudi Poepel Lemaitre

based on material from Volker Markl, Alexander Alexandrov, Jonas Traub, Martin Kiefer

Basic system architecture



Where are we today?



Last Lecture

- Operators to implement nested loop joins and additional predicates in the WHERE clause.
- NestedLoopJoinOperator
 - Nested loop join
 - Indexed nested loop join
- IndexLookupOperator
 - Retrieve the RID of a matching tuple based on a predicate.
- IndexCorrelatedLookupOperator
 - Retrieve the RID of a matching tuple based on a correlated tuple.
- FetchOperator
 - Fetch the columns of a tuple retrieved through an index.
- FilterOperator
 - Filter tuples based on a predicate.
- FilterCorrelatedOperator
 - Filter tuples based on a correlated tuple.

This Exercise

- Implementing sort-based operators:
 - `SortOperator`
 - `GroupByOperator`
 - `MergeJoinOperator`

Sort Based Operators

Sort Operator

- Sorts tuples in a table based on an attribute or set of attributes.
- Provides order as requested in the **ORDER BY** clause.
- Provides order for grouping (**GROUP BY**) & aggregation (**AVG**, **SUM**, **MIN**, **MAX**, **COUNT**...).
- Provides order for sort-based joining (**MergeJoin**).

Two-Phase Multiway-Merge-Sort

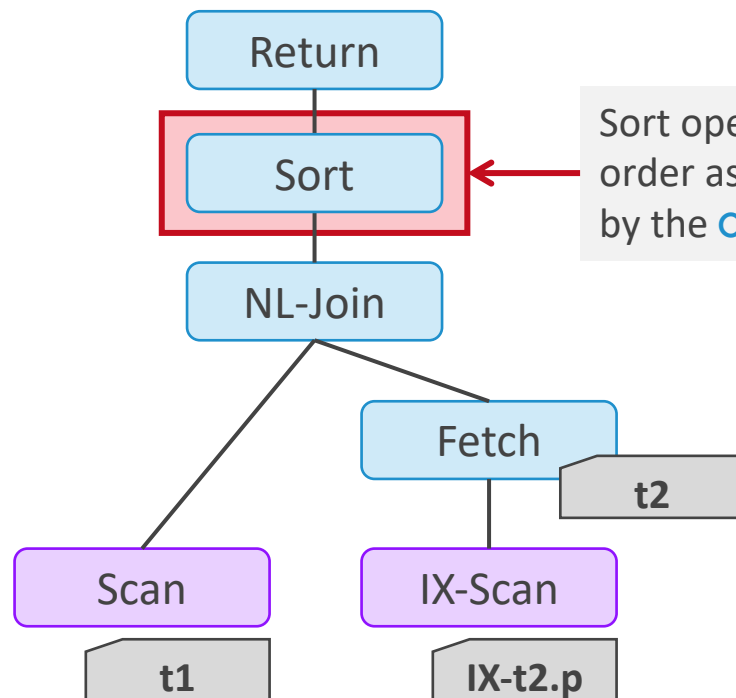
- Basic sort operator
- In-Memory while enough memory is available
- External, when memory is not sufficient
- Works roughly while $B(R) < M^2$

Sort-based Operators

- Merge Join Operator
 - Join two sorted inputs on the join attributes.
- Group By Operator
 - Expects the input stream to be sorted by the grouping columns.
 - Single pass over the stream aggregating always only the current group
 - Result still sorted by the grouping columns

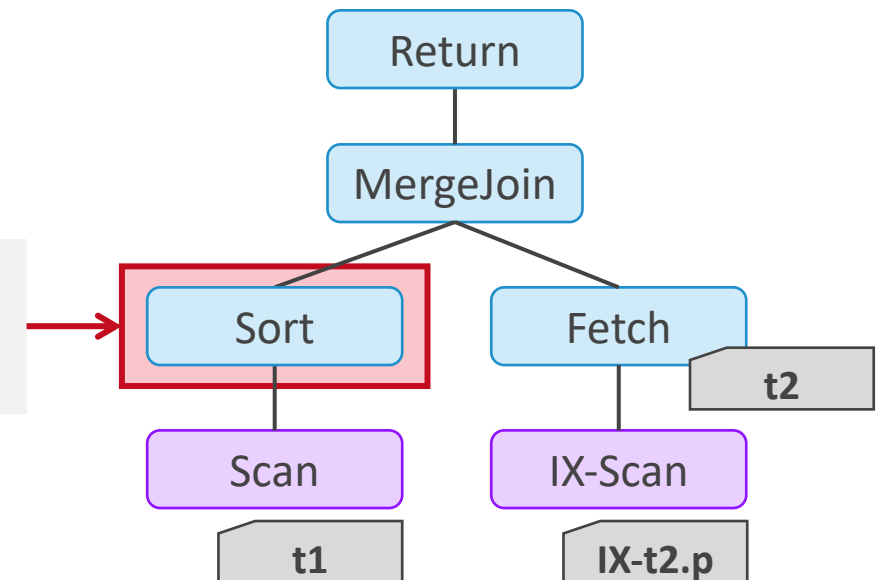
Example: Plans with Sorting I

```
SELECT t1.a, t2.b
FROM tab1 t1, tab2 t2
WHERE t1.f = t2.p
ORDER BY t1.f
```

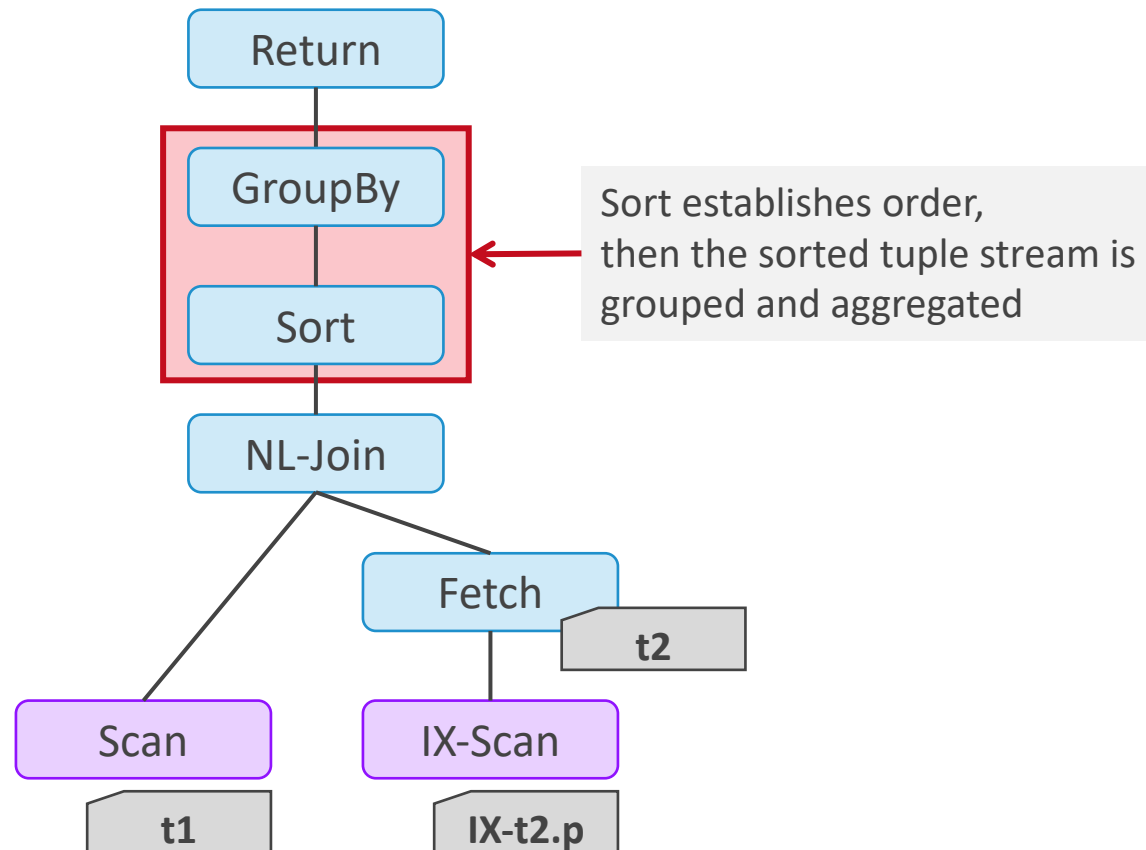


Sort operator providing order as required by the **ORDER BY** clause

Sort providing order for merge join. Order will be preserved and fulfills the requirement of the **ORDER BY** clause.



Example: Plans with Sorting II

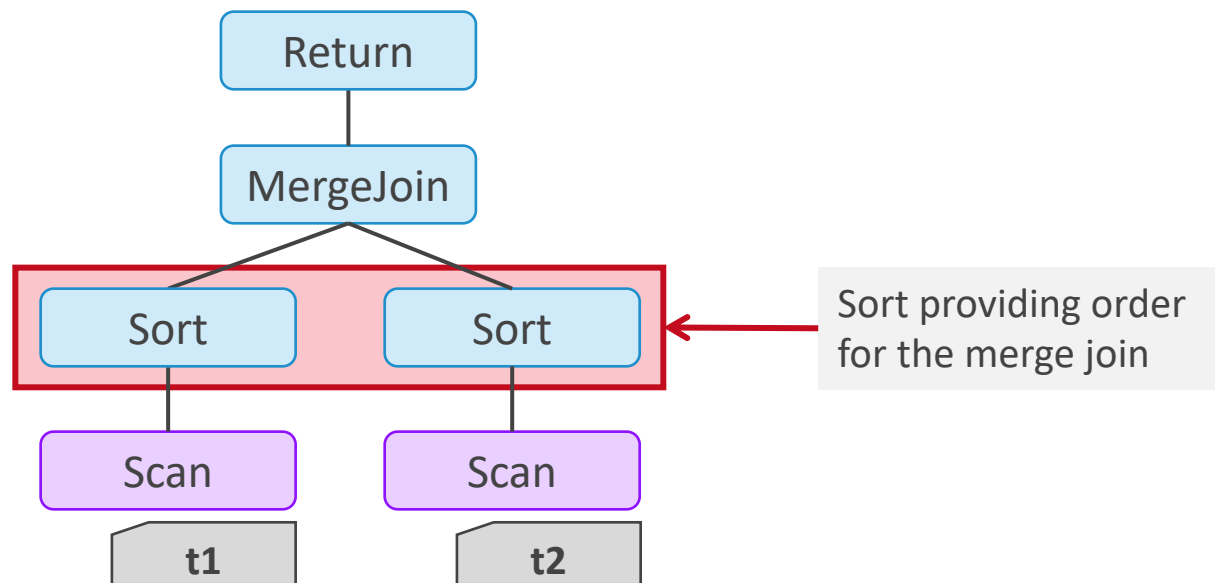


```

SELECT t1.a, AVG(t2.b)
FROM tab1 t1, tab2 t2
WHERE t1.f = t2.p
GROUP BY t1.a

```

Example: Plans with Sorting III



```

SELECT t1.a, t2.b
FROM tab1 t1, tab2 t2
WHERE t1.f = t2.p
  
```

Two-Phase Multiway Merge Sort

Two-Phase Multiway Merge Sort

- Phase 1: Production of the sorted sub lists.
- Phase 2: Merging of the sorted sub lists.
- Correlation: Not evaluated, passed on to the child.

Query Heap

- Temporary area to store intermediate results
 - Temporary area in memory to sort a sublist
 - Temporary area in storage to store sorted sublists
- Query Heap often also called SortHeap, TempHeap, ...
- Provides space for Sorts, HashTables and TemporaryTables...
- In our mini-dbs: `de.tuberlin.dima.minidb.qexec.heap.QueryHeap`

Phase 1: Initialization

- Create a Query Heap for the current operation
 - **HINT:** Method `QueryHeap#reserveSortHeap()` returns `sortHeapID`
- Get a chunk of memory from the heap to use as the sort buffer.
 - This is the current sublist
 - **HINT:** Methods `QueryHeap#getSortArray()` and `getMaximalTuplesForInternalSort()` methods
 - Use the `sortHeapID` to identify the query heap for the operation

Phase 1: Sorting

- Get tuples from the child operator and add them to the current sublist array until the child operator is exhausted or the list is full.
- Sort the sublist with an in-memory sort algorithm.
 - Typically: Quicksort, HeapSort, RadixSort, TreeSort
 - **HINT:** Use the algorithm provided by the JVM: `Arrays.sort`
- If the sublist is full and sorted, and the child operator **is not exhausted**, call the QueryHeap to write the sublist to secondary storage.
 - **HINT:** use method `QueryHeap#writeTupleSequenceToTemp()`

Multiway Merge Sort Phase 2

- Merge the sorted sublists, always keeping the first k blocks from each sublist in memory
 - $k = M / s$, where M = number of blocks that the sort gets assigned from the QueryHeap, s = number of sorted sublists.
 - **HINT:** The method `QueryHeap#getExternalSortedLists` will take care of this!

Phase 2: Initialization

- Create an iterator for each sorted sublist
- **HINT:** Method `QueryHeap#getExternalSortedLists`

Phase 2: Merging

- Look at each sublist and return the smallest element
 - If multiple blocks have the same sort key, then take the element from the block belonging to the earliest produced sub-list.
→ Makes the sort stable.
- Advance the iterator of the sublist from which the element was returned

Working with the Query Heap API

- Query Heap is used to control the amount of memory used by operators that accumulate intermediate results (Sort, HashJoin).
 - Ensure that the memory consumption stays below a limit.
 - Ensure that multiple concurrent Sorts or HashJoins have enough memory.
- A sort operator needs to reserve space on the query heap (in the `open()` method):
 - For this, use the method `int reserveSortHeap(ColumnSchema[], tupleSchema, int estimatedCardinality)`
 - Returns an ID that is used to identify that operator's portion of the heap.
 - The tuple schema is passed to let the heap estimate how many tuples should be kept in memory.
- After completion (in the `close()` method), the operator releases its portion of the heap
 - For this, use the method `releaseSortHeap(int heapId)`
 - Must be done in any case (even if the operator fails!).

Working with the Query Heap API

Merge Sort Phase I:

- Get an array for sorting from the heap.
 - Arrays are pooled and reused, to get one use the method `DataTuple[] getSortArray(int heapId)`
 - To get maximum number of tuples that fit in a sublist call `int getMaximalTuplesForInternalSort(int operatorId)`
- Sorted sublists may be written to Tempspace
 - `void writeTupleSequencetoTemp(int heapId, DataTuple[] tuples, int numTuples)`
- For external sorts: Sort array should be released before phase 2
 - `void releaseSortArray(int heapId)`

Working with the Query Heap API

Merge Sort Phase II:

- Get iterators over the sorted sublists.
 - Iterators will load the blocks from the disk as they are needed to keep memory consumption low.
 - `ExternalListIterator[] getExternalSortedLists(int heapId)`

Merge Join

Merge Join

- Requires inputs to be sorted on all key columns **in the same manner!**
 - Provided by sort operators below, **not part of the join operator.**
- Go over both inputs in a *zig-zag* fashion.
- Then, form the cartesian product or the subsets of tuples with identical keys.
- Correlation: Not evaluated, passed on to both children.

Merge Join Hints

- Merge join operator should be rather stateless → Should not buffer many tuples
- Flexible code required to handle the cases when a single join key occurs:
 - Once on both inputs.
 - Once in the left input, multiple times in the right.
 - Multiple times in the left input, once in the right.
 - Multiple times in both inputs.
- What to do when both sides have way too many tuples with the same join key?
 - Extreme case: Join two relations with 1 million tuples each that all have the same join key.
(you **don't** need to cover this case)

Group By and Aggregates

Group By Operator (Aggregates)

- Requires tuples sorted on all grouping columns.
 - Provided by sort operators below, not part of the group by operator.
- Processes the tuple stream from the child operator:
 - Remember the desired output is one tuple of grouping columns and the current aggregate for each aggregated column.
 - If the grouping columns have the same value, aggregate the next tuples columns with the current aggregates.
 - If the grouping columns change:
 - Current tuple can be composed and returned.
 - A new set of aggregates can be started.
- Correlation: Not evaluated, passed on to the child.

Some Hints on the GroupBy Operator

- The set of grouping columns may be empty (e.g., if you count or sum over everything).
- NULL values are ignored by the aggregation functions (NULL cannot be arithmetic or has no natural order among the other elements).
- NULL is considered in grouping: It is possible to have groups.
 - (val1, val2, val3), (val1, val2, NULL), (val1, NULL, val2), and so on...
- COUNT, MIN, and MAX apply to all types.
 - use `DataField#compareTo()` to evaluate min and max.
- SUM, AVG only applicable to types that implement **ArithmeticType**
 - SmallInt, Int, BigInt, Float, Double
 - Check if a field type implements **ArithmeticType** by using the method `BasicType#isArithmeticType()`.

Some Hints on the GroupBy Operator

- If the child of the group by operator does not produce any tuples (the first tuple is immediately null), then the behavior depends on the following:
 - If there are grouping columns, the operator returns null (produces no tuple).
 - If there are only aggregation columns, then the operator returns a tuple with these aggregation columns. The aggregates are aggregates over nothing, which means:
 - COUNT = 0, SUM = 0, MIN = NULL (not applicable), MAX = NULL, AVG = NULL (div by zero?)
- The data type of an aggregate column in the produced tuple is the same as that of the aggregated column.
 - Exception: COUNT always produces an IntField
 - Make sure to return '0' or 'NULL' as the correct type (SmallIntField, DoubleField, ...).
- If we have only aggregation functions and no grouping (i.e. everything is one group), then we need no sort.

Some Hints on the GroupBy Operator

- Have a look at the interface `de.tuberlin.dima.minidb.core.ArithmeticType`
 - Implemented by the types that apply to SUM and AVG.
 - Contains function to produce zero.

```
DataField f = ...  
ArithmeticType<DataField> a = DataType.asArithmeticType(f);  
a.divideBy(6);
```

```
// get Zero for a data type and add some value  
DataType type = schema.getColumnType(x); // assume data type is float  
ArithmeticType<DataField> zero = type.createArithmeticZero();  
  
// zero will be of type FloatField  
zero.add(...);  
zero.multiplyWith(...);
```

Exercise 6

Exercise 6

- Implement 3 operators using the following interfaces:
 - `de.tuberlin.dima.minidb.qexec.SortOperator`
 - `de.tuberlin.dima.minidb.qexec.MergeJoinOperator`
 - `de.tuberlin.dima.minidb.qexec.GroupByOperator`
- Implement the following factory methods:
 - `createSortOperator`
 - `createMergeJoinOperator`
 - `createGroupByOperator`

Tests and points

Test name	Points	Description
testInternalSort	2	Tests sorting a table that fits in memory.
testExternalSort	4	Tests sorting a table that doesn't fit in memory.
testSimpleCount	1	Test a simple count aggregate without grouping attributes.
testGroupCustomers	2	Test various aggregates with grouping attributes.
testGroupCustomersEmpty	1	Test various aggregates with grouping attributes on an empty table.
testAggregateCustomersEmpty	1	Test various aggregates without grouping attributes on an empty table.
testMergeJoin	3	Tests the merge join.